Monads Explained

Izaak Weiss

1 What are Monads?

It is entirely reasonable that this is the first question that anyone learning Monads asks, and it is also entirely reasonable that anyone who is teaching Monads tries to answer it. However, Monads are a complex concept that cannot be explained in a single sentence or even a single paragraph; to understand monads you must simultaneously understand the problem they are trying to solve, the implementation and the interface for working with them, and the theoretical computational background. Therefore, I will not try and answer this question in a single phrase; my explanation of what Monads are is the entirety of this paper.

2 The Our First Monads

In this section, we'll explore a common problem surrounding how to report errors to the caller of a function.

2.1 Division

```python
def division(x, y):
    return x / y
```

Consider the above code fragment. This is a very simple function; one that is so simple it hardly deserves to exist. However, this function, despite it's simplicity, can throw an error if it's inputs are badly formed. `division(1.0,0.0)` will throw a `ZeroDivisionError: division by zero`. It's possible that we want to recover from this error gracefully; check the inputs and see if an error will occur, and return some error code instead of raising an exception.

However, we must decide what the error code should be. We can't choose `0.0`, because that is correctly returned by `division(0.0,1.0)`. We can't choose `-1.0`, because that is correctly returned by `division(-1.0,1.0)`. In fact, this function can return any possible floating point number, so we can't choose a floating point number as our error code. Luckily, python is dynamically typed, so we can return a float on success, but a nonetype on failure.

```
def division(x, y):
    if y == 0:
        return None
    return x / y
```

Now we've written a function that checks whether or not division is possible, and performs division if it is, but returns an error code if it is not.

2.2 Indexing

```
def index(ls, i):
    return ls[i]
```

The above code is similar to the last example; it will perform an index lookup into a list, and return the item from the list if it can. If it can't, we're still left with the problem that it throws an exception. `index([1,2,3],10)` throws an `IndexError: list index out of range`. Let's try and do the same thing as above; rewrite this function so that it doesn't throw an error, but instead uses an error code to signal something has gone wrong.

Our first guess might be to have our error code be the same as above, and just return `None`. However, this isn't possible, because the code `index([None], 0)` would also return `None`, making it impossible to tell in practice if we have an error or not. In fact, python lets any value be inside of a list; there is no possible error code we can return that can't also be in the list. Luckily again, python lets us use multiple return values.

```
def index(ls, i):
    if i < 0 or i >= len(ls):
        return True, None
    return False, ls[i]
```

This lets us write code like the below snippet, and fixes the problem of python being able to have anything we want in a list.

```
failed, value = index([1,2,3],0)
if failed:
    print("Oh no, we failed")
else:
    print(value)
```

2.3 Combining The Above

```python
def divide_elements(ls, i1, i2):
    return ls[i1]/ls[i2]
```

Now, we've combined the two operations in python which might lead to an exception, and we've done it in a way that allows for three different operations to result in an exception. We can use the above functions to rewrite this function so that it won't throw any errors, by checking after each step for an error code.

```python
def index(ls, i):
    if i < 0 or i >= len(ls):
        return True, None
    return False, ls[i]

def division(x, y):
    if y == 0:
        return None
    return x / y

def divide_elements(ls, i1, i2):
    failure, v1 = index(ls, i1)
    if failure:
        return None

    failure, v2 = index(ls, i2)
    if failure:
        return None

    return division(v1, v2)
```

This code works nicely; you can throw two types of errors at it, and it returns `None` when either error would occur. `divide_elements([1,2,3,0],1,10)` and `divide_elements([1,2,3,0],1,3)` both return `None`, and `divide_elements([1,2,3,0],1,2)` returns `0.6666666666666666`.

This is nothing special so far. All we've done is rewritten a piece of code so that it doesn't throw an exception. In fact, we've probably made the code worse; python is built around dynamic types and exception throwing, so the more pythonic way of writing the above code is more like the following.

```python
def divide_elements(ls, i1, i2):
    try:
        return ls[i1]/ls[i2]
    except (IndexError, ZeroDivisionError):
        return None
```

However, consider these problems in a language like C. No exceptions, no dynamic return types, and no returning multiple values. This isn't to say the problem can't be solved. C programmers have their own ways of getting around this sort of problem. But different languages have different requirements, and there are many ways to solve a problem. Next up I'm going to introduce our first monad, a monad that helps deal with the problem we faced above - it's not the only way, but it's an elegant way that works in statically types languages without exceptions.

2.4 Our First Monad

The Option Monad, also called the Maybe Monad in many programming languages, is a way of representing the result of a function, the result of any computation, that might result in an error and produce no meaningful output. We call an Option Monad that has a value and represents a successful computation Some, and we call one that doesn't have a value and represents a failed computation None.

These are really easy to write in programming languages like Haskell, Scala, or Rust, but I'm going to write it in Python to help avoid any confusion about the inner workings of the Monad. In the end, all Monads are just objects, like trees, lists, or dictionaries.

```python
class OptionException(Exception):
    pass

class Option:
    def __init__(self, failed, value):
        self._failed = failed
        self._value = value

    def __repr__(self):
        if self._failed:
            return 'Option.none()'
        else:
            return 'Option.some({})'.format(self._value)

    def __str__(self):
        if self._failed:
            return 'None'
        else:
            return 'Some({})'.format(self._value)

    def is_some(self):
        if self._failed:
            return False
        return True
    def is_none(self):
        return not self.is_some()

    def unwrap(self):
        if self._failed:
            raise OptionException('This Option has no value')
        else:
            return self._value

    @classmethod
    def some(cls, x):
        return cls(False, x)

    @classmethod
    def none(cls):
        return cls(True, None)
```

This is the longest piece of code we've had so far, so let me break it down bit by bit.

```
class OptionException(Exception):
    pass
```

This is how you create a custom exception in python. I could have used a built in error if I wanted, but I wanted it to be clear that this was part of the Option Monad that was causing the exception and not something else. You probably should object at this point, and yell at me, saying something like, "isn't the whole point of the Option monad to get rid of exceptions?" Well, if you use the Option Monad correctly, you'll never run into this exception.

```
class Option:
    def __init__(self, failed, value):
        self._failed = failed
        self._value = value
```

The `__init__` function is the constructor or initializer function for classes in python. Here all we do is create a boolean that indicates whether or not we have failed the computation or not, and if we haven't failed it, we store the result of the computation in `_value`. Note that if `_failed` is `True`, then we don't care what is in `_value`, because the computation has failed and that value has no meaning. I will note that the users of this class will probably never call `__init__` themselves.

```
    def __repr__(self):
        if self._failed:
            return 'Option.none()'
        else:
            return 'Option.some({})'.format(self._value)
    def __str__(self):
        if self._failed:
            return 'None'
        else:
            return 'Some({})'.format(self._value)
```

This is how you define string representations of values in python. `__str__` is called when you cast something to a string or print it, and `__repr__` is shown in the python REPL, and is supposed to be a little more informative. But these are mostly for ease of use; they don't really matter when it comes to Monads in general.

```
    def is_some(self):
        if self._failed:
            return False
        return True

    def is_none(self):
        return not self.is_some()

    def unwrap(self):
        if self._failed:
            raise OptionException('This Option has no value')
        else:
            return self._value
```

These three functions are the meat of the Option Monad; these are the ways we interact with it. The `is_some` function returns `True` when there is a meaningful return value, and `False` if the computation failed. `is_none` does the opposite. `unwrap` returns the value of the Option Monad *if there is a value to be returned*, otherwise it throws an error. In order to use the `unwrap` function without an error, you must first check and see whether the computation succeeded or failed.

```
    @classmethod
    def some(cls, x):
        return cls(False, x)

    @classmethod
    def none(cls):
        return cls(True, None)
```

These functions, decorated with `@classmethod`, aren't methods of the object. Instead, they're methods that exist as part of the class itself; here, we use them as alternate constructors.

At this point, let me rewrite our exception free code from above using the Option Monad.

```
def division(x, y):
    if y == 0:
        return Option.none()
    return Option.some(x / y)

def index(ls, i):
    if i < 0 or i >= len(ls):
        return Option.none()
    return Option.some(ls[i])

def divide_elements(ls, i1, i2):
    res1 = index(ls, i1)
    if res1.is_none():
        return Option.none()

    res2 = index(ls, i2)
    if res2.is_none():
        return Option.none()

    return division(res1.unwrap(), res2.unwrap())
```

The above code is the exact same length in lines; and already has some benefits. One, every function has exactly one return type - not so useful in python, but I hope you can see why they'd be useful in statically typed languages. Two, we don't have to remember the convention for every function. It's not a mess of trying to remember that `division` returned `None` for an error, but `index` returned `False, None` for an error. But in the end, it's not that much of an improvement.

And that's because we haven't yet implemented the most important function for a Monad. This is the most important but also the most complicated part of the Option Monad, so I am going to insert a header for this.

2.5 The 'Bind' Function

```
    def bind(self, function):
        if self.is_none():
            return type(self).none()

        val = self.unwrap()
        return function(val)
```

The bind function first checks whether or not `self` is None or Some. If `self` is None, then we return a new None instance. If self is Some, we unwrap the value, and pass it into the function, returning what the function returns. Because we're sort of pretending to be in a typed language here, we should remember that the function passed in *should* return an Option Monad.

So, with `bind` in mind, let's consider the our code again.

```python
def division(x, y):
    if y == 0:
        return Option.none()
    return Option.some(x / y)

def index(ls, i):
    if i < 0 or i >= len(ls):
        return Option.none()
    return Option.some(ls[i])

def divide_elements(ls, i1, i2):
    res1 = index(ls, i1)
    res2 = index(ls, i2)

    partial = lambda x: lambda y: division(x,y)

    return res2.bind(res1.bind(partial))
```

I've changed a few things, so go back and look at what I've done. There's this new, weird line `partial = lambda x: lambda y: division(x,y)`. This is a way of defining an inline function in python; here, I have a function of x that returns a function of y, that itself returns `division(x,y)`. That's a bit complicated, so think about it for a second.

What I then do on the return statement line is I use bind to apply the function to these arguments; the function `division` will fail drastically if it gets an Option Monad in as an argument; it would fail on the comparison to zero and it would also fail on trying to divide two Option Monads. By using `bind`, I am telling the Options to apply the function to themselves if they have a value, or returning None if they don't have a value. This wraps the error handling into the data structure that represents errors without me having to ever write an explicit `if x.is_some():` check.

You may still think this sort of thing is useless; and in python, for such a simple example, it kinda is! But as we continue to explore Monads, we will encounter some weirder and wilder examples of things that Monads make simpler.

2.6 A More Complex Example

In order to give a more illustrative example of where the Option Monad can be more useful, consider the following problem; open a file, and read the first whitespace separated word from the beginning of the file, and parse it into an integer if possible. This problem is fairly easy to do with built in python functions, but the Option monad would make it easier. However, none of python's built in functions use the Option monad, so we will have to rewrite them so that they do. In languages with the Option monad as a star player, this isn't an issue.

```python
def option_open(filename, mode='r'):
    try:
        fd = Option.some(open(filename, mode=mode))
    except Exception:
        fd = Option.none()
    return fd

def option_read(fd, size=-1):
    try:
        data = Option.some(fd.read(size))
    except Exception:
        data = Option.none()
    return data

import re

def option_match(pattern, string):
    match = re.match(pattern, string)
    if match:
        return Option.some(match)
    else:
        return Option.none()

def option_get_group(match, group):
    try:
        g = match.group(group)
    except Exception:
        g = None

    if g == None:
        return Option.none()
    else:
        return Option.some(g)

def option_int(s):
    try:
        i = Option.some(int(s))
    except Exception:
        i = Option.none()
    return i
```

I'm not going to explain these functions too much; basically they do the same thing as their built in python counterparts, but they return Some if the computation succeeds and None if it fails. This will allow us to use `bind` to chain these functions.

```
result = (Option.some('text.txt')
.bind(option_open)
.bind(option_read)
.bind(lambda x: option_match(r'\s*(\S*)', x))
.bind(lambda x: option_get_group(x, 1))
.bind(option_int))
```

This code is useful because if an error happens at any time during the computation, it just passes a None Option through the rest of the bind functions.

We can make this look cooler by choosing an infix operator to overload. By convention, `>>=` is used, but that's hard to to do in python, so we are going to use `>>`. We can override that operator in python with the following code:

```
def __rshift__(self, function):
    return self.bind(function)
```

This code let's us rewrite the above option code as the following.

```
result = (
    Option.some('text.txt')
      >> option_open
      >> option_read
      >> (lambda x: option_match(r'\s*(\S*)', x))
      >> (lambda x: option_get_group(x, 1))
      >> option_int
    )
```

2.7 The Result Monad

Some of you may be thinking about the usefulness of error messages. When the above code is run, you can't tell where or why an error occurs. However, these is a simple modification to the Option Monad that makes is able to convey error messages. Some languages call this the Result Monad. I'm not going to go in depth as to how it works, because it's very similar to the Option Monad. I will, however, highlight one function from below.

```
    def recover(self, function):
        if self.is_error():
            return function()

        return self
```

The `recover` function is the opposite of `bind`; if the result is a failure, it will attempt to recover computation by replacing the current Result Monad with the result of whatever function is passed in. If the result is a success, however, it leaves the value alone. This is a nonstandard addition to monads; I find it useful, and so I've added it, but it might not exist in every language, hence why I haven't brought it up yet. I will, however, use it later.

```python
class ResultException(Exception):
    pass

class Result:
    def __init__(self, failed, value, message):
        self._failed = failed
        self._message = message
        self._value = value

    def __repr__(self):
        if self._failed:
            return 'Option.error({})'.format(repr(self._message))
        else:
            return 'Option.ok({})'.format(repr(self._value))

    def __str__(self):
        if self._failed:
            return 'Error({})'.format(self._message)
        else:
            return 'Ok({})'.format(self._value)

    def is_ok(self):
        if self._failed:
            return False
        return True

    def is_error(self):
        return not self.is_ok()

    def unwrap(self):
        if self.is_ok():
            return self._value
        else:
            raise ResultException('This Result is an Error')

    def error_msg(self):
        if self.is_error():
            return self._message
        else:
            raise ResultException('This Result is Ok')

    def bind(self, function):
        if self.is_error():
            return self

        val = self.unwrap()
        return function(val)

    def recover(self, function):
        if self.is_error():
            return function()

        return self

    def __irshift__(self, function):
        return self.bind(function)

    @classmethod
    def ok(cls, val):
        return cls(False, val, None)

    @classmethod
    def error(cls, msg):
        return cls(True, None, msg)

# The following are built in functions rewritten to work with the Result Monad

def result_open(filename, mode='r'):
    try:
        fd = Result.ok(open(filename, mode=mode))
    except Exception:
        fd = Result.error("Failed to open the file")
    return fd

def result_read(fd, size=-1):
    try:
        data = Result.ok(fd.read(size))
    except Exception:
        data = Result.error("Failed to read from the file")
    return data

import re

def result_match(pattern, string):
    match = re.match(pattern, string)
    if match:
        return Result.ok(match)
    else:
        return Result.error("Failed to match the pattern")

def result_get_group(match, group):
    try:
        g = match.group(group)
    except Exception:
        g = None

    if g == None:
        return Result.error("Failed to get the group from the match")
    else:
        return Result.ok(g)

def result_int(s):
    try:
        i = Result.ok(int(s))
    except Exception:
        i = Result.error("Failed to parse into an integer")
    return i


result = (
    Result.ok('text.txt')
      >> result_open
      >> result_read
      >> (lambda x: result_match(r'\s*(\S*)', x))
      >> (lambda x: result_get_group(x, 1))
      >> result_int
    )

print(result)
```

2.8 Either Monad

Another common monad that is absolutely useless in python but invaluable in other languages, and is in fact a more general form of the Result Monad, is the Either Monad. An Either Monad is really useful in strictly typed languages, because it can be returned from a function that can return two possible types.

I'm not going to implement this in python, because this is thoroughly useless for a python program, but it's essentially identical to the Result Monad implementation. Try it out!

3 A Parsing Monad

This is our first look into a complex and powerful Monad that represents a very useful computer science tool. We are going to use a Monad called the Parsing Combinator, which basically recreates and improves upon regex using Monads. Here we see the truth of the oft said but little understood aphorism that Monads represent computation; here, they represent the computation of regular expressions.

It's also worth noting that there is a python Parsing Combinator library on the python package index, so if you want to use this for real, use their code and not mine; it's much more complete and probably nicer to use.[1]

3.1 The Code

First of all, I am going to use the above Result Monad to help me implement this Parsing Combinator. Even though the parsing itself is Monadic, it has to return a result that can either be a success or a failure; so the Result Monad is going to be useful.

With that said, let's step through the code for our Parsing Combinator.

```
class Parser:
    def __init__(self, function):
        self._function = function

    def __call__(self, text, index):
        return self._function(text, index)

    def __repr__(self):
        return '<Parsing Combinator>'
```

The basic idea behind our Parsing Combinator is that the Monad holds a function that takes a text and an index, and starts looking at the text from that index, and returns either an Error Result, in which case Parsing has failed, or it returns the text and a new index to start at. In order to make this a bit easier for us, we will implement the special method `__call__`, so that we can write stuff like `self(text, index)` instead of `self._function(text, index)` in the future. It's really hard to print functions meaningfully, so we're just going to define our representation as a constant string.

```
    def bind(self, function):

        def bind_func(text, index):
            return self(text, index).bind(lambda x: function(*x))
```

[1] https://pypi.python.org/pypi/parsec

```
        return Parser(bind_func)
```

`bind` is, of course, the star of the show. In order to bind, we need to return a new Parser, and the Parser needs to hold a function. In order to do this, we will define an 'inner function' here, also called a 'closure'. The 'closure' will take a text and an index, and call the current Parser's function on the those. That will result in a result variable, which we will bind the function to.

```
    def compose(self, other):
        return self.bind(lambda x: other(*x))

    def choice(self,other):

        def choice_func(text, index):
            return self(text, index).recover(lambda: other(text, index))

        return Parser(choice_func)

    def many(self):

        def repeat_func(text, index):
            return self(text, index).bind(lambda x:
                repeat_func(*x)).recover(lambda:
                    Result.ok((text, index)
                )
            )

        return Parser(repeat_func)
```

`compose` is a simple enough function that we can write it just using the bind operation of the Parser monad. All it does is first attempt to parse using self, and then if it succeeds, it continues parsing where self ended with other.

`choice` requires a bit more complexity; we first try to apply self to the text, and if that fails, we then apply other starting from the same spot. Be careful with choice though; it only implements local choice. If you ask it to choose between **x** and **y**, and **x** exists but causes the parser to fail later on, it won't backtrack and try **y**. It will only try **y** if parsing **x** fails.

`many` is the most complex; it takes a parser and repeats parsing with it, recursively, until it fails, at which point it returns that last success. I do this recursively, in one expression, because that's the sort of power and freedom that monads give us; but you can expand it if you want to a more familiar form.

Below, I create symbolic aliases for some of the above functions, for ease of use.

```
    def __rshift__(self, function):
        return self.bind(function)
    def __add__(self, other):
        return self.compose(other)

    def __or__(self, other):
        return self.choice(other)
```

Now, all of these aren't special with regards to it being a monad; we could implement these functions completely outside of the class; these are like implementing a factorial function inside of an integer class. We're mostly putting them here because it's easier to process in written code as `parser1 | parser2.many()` than `choice(parser1, many(parser2))`. However, it may take some getting used to.

Next, I'm going to create some constructors for these Parsers; they're going to be easy ways to create simple, atomic Parsers, which we will then combine with the above function to make a complex Parser Combinator.

```
    @classmethod
    def char(cls, val):

        def match_char(text, index):
            try:
                current = text[index]
            except IndexError:
                return Result.error('End of String encountered')

            if current == val:
                return Result.ok((text, index+1))
            else:
                return Result.error('Failed to match character {}'
                .format(repr(val)))

        return cls(match_char)
```

This above function takes a character value, and creates a function that matches the current index with that character, and puts that function into a Parser. For example, `Parser.char('a')` is a parser that matches exactly one `'a'`, and nothing more.

```
    @classmethod
    def oneof(cls, charls):

        def match_charls(text, index):
            try:
                current = text[index]
            except IndexError:
                return Result.error('End of String encountered')
            if current in charls:
                return Result.ok((text, index+1))
            else:
                return Result.error('Failed to match one of {}'.format(list(charls)))

        return cls(match_charls)
```

This function matches any one of a bunch of characters. Obviously, we could take a bunch of single character parsers and combine them using `choice`, it's nicer to be able to create a parser with one command.

```
    @classmethod
    def noneof(cls, cha):

        def none_charls(text, index):

            try:
                current = text[index]
            except IndexError:
                return Result.error('End of String encountered')

            if current not in charls:
                return Result.ok((text, index+1))
            else:
                return Result.error('Found one of when there shouldn\'t be {}'.format(list(c

        return cls(none_charls)
```

This function does the exact opposite of `oneof`; it checks and makes sure that the current character isn't any character that you passed into it.

```
    @classmethod
    def empty(cls):

        def match_empty(text, index):
            return Result.ok((text, index))

        return cls(match_empty)
```

Finally, we have a Parser that matches the empty string. It doesn't increment the index, and it always returns a success, so it does nothing on it's own; however, it is useful in combination with other things.

```
    @classmethod
    def parse_prefix(cls, parser, string):

        def split(tup):
            text = tup[0]
            index = tup[1]
            return Result.ok((text[:index], text[index:]))

        return parser(string, 0) >> split

    @classmethod
    def parse_total(cls, parser, string):

        def split(tup):
            text = tup[0]
            index = tup[1]
            return Result.ok((text[:index], text[index:]))

        def check_full(tup):
            if tup[1] == '':
                return Result.ok(tup[0])
            else:
                return Result.error("The match did not consist of the entire string: {} was

        return parser(string, 0) >> split >> check_full
```

These two functions help make it easier to use the parser; instead of calling the parser ourself, we use these functions to parse either a prefix of the string using our parser, or the entire string.

3.2 Using the Parser Combinator

Now we have a powerful enough Parser Monad to recreate all of the flexibility and power of regular expressions. We can combine parsers to make more complex ones, we can define functions that return parsers that recreate common regular expressions syntax, so let's try parsing something simple; let's try and write a Parser Combinator that parses numbers.

A number can be as simple as `12` or as complex as `12.00123e45`, so we're going to need to build up a complex parser. Let's start with creating a parser that parses 1 or more of consecutive digits. (Remember that `many` is a zero or more parser!)

```
digits = Parser.oneof('0123456789') + Parser.oneof('0123456789').many()
```

In fact, the above code gives us the blueprint for a more general one or more function (which I'm going to add to the Parser class for ease of use).

```
def many1(self):
    return self + self.many()

digits = Parser.oneof('0123456789').many1()
```

Now, we need to express an optional decimal place, followed by one of more digits.

```
decimal = Parser.char('.') + digits | Parser.empty()
```

This gives us the blueprint for a more general optional parser, so I'm going to add that to our parser class as well.

```
def optional(self):
    return self | Parser.empty()

decimal = (Parser.char('.') + digits).optional()
```

Now, we need an exponent part, which is pretty simple given the above. However, we do need one additional component, a sign.

```
sign = (Parser.char('+') | Parser.char('-')).optional()
exponent = ((Parser.char('e') | Parser.char('E')) + sign + digits).optional()
```

Finally, we can put these three together to get:

```
number = sign + digits + decimal + exponent
```

Here's our finished code, and it's results on some possible inputs:

```
digits = Parser.oneof('0123456789').many1()
decimal = (Parser.char('.') + digits).optional()
exponent = ((Parser.char('e') | Parser.char('E')) + digits).optional()
number = digits + decimal + exponent

Parser.parse_total(number, '12')
    # Ok(12)
Parser.parse_total(number, '12e10')
    # Ok(12e10)
Parser.parse_total(number, '2.12345e100')
    # Ok(2.12345e100)
Parser.parse_total(number, 'hello world')
    # Error(Failed to match one of
    # ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'])
Parser.parse_total(number, '99 bottles of beer on the wall')
    # Error(The match did not consist of the entire string:
    # ' bottles of beer on the wall' was left over)
Parser.parse_total(number, '')
    # Error(End of String encountered)
```

Furthermore, we can use a result-oriented version of python's `float` function to cast these results to be floats instead of strings.

```
def result_float(x):
    try:
        return Result.ok(float(x))
    except Exception:
        return Result.error('Failed to cast to a float')

Parser.parse_total(number, '7.22345e10') >> result_float
    # Ok(72234500000.0)
```

3.3 The Really Cool Part

I have to figure out how to do this in python still, but it's so cool in Haskell.

4 Theory Time

Now that we've seen many types of Monads, we should discuss the technical, boring discussion of what a Monad is. A Monad is, in essence, any construct, usually an object, in a language that satisfies the following criteria. I will say each criterion in two ways; a simple way, and a correct way.

4.1 The Monad Laws

- For any type `t`, and Monad type `M`, `M t` is the type of the Monad holding that type.
- *Monads can hold values*

- There is a function called 'unit' or 'return' that has type `t -> M t` which injects a value of type `t` into a Monad of type `M t` in some simple way.
- *There is a function to create Monads from regular values*

- There is a binding operation of type `M t -> (t -> M u) -> M u` which maps the value of a Monad of type `M t` into another Monad of type `M u` by using a function that maps a value of type `t` into a Monad of type `M u`
- *There is a bind function, as discussed earlier*

These above operations also have to follow a certain set of laws. These laws are super simple; basically they just exist to make sure that the bind function and the constructor don't do anything funky.

- The unit function acts as a neutral element of bind
- *Calling bind on the unit function (or constructor) doesn't do anything*

Example:

```
Option.some(5).bind(Option.some) == Option.some(5)
```

- Binding two functions in series is the same as binding the result of composing those two functions
- *You don't have to worry about binding doing weird things to your values; an example is really useful here*

```
def one_over(x):
    if x == 0:
         return Option.none()
    return Option.some(1/x)

def two_over(x):
    if x == 0:
        return Option.none()
    return Option.some(2/x)

Option.some(5).bind(one_over).bind(two_over) == Option.some(5) \
    .bind(lambda x: one_over(x).bind(two_over))
```

4.2 An Alternate Theory

If you want, you can replace the bind function with two slightly different functions, and you get the same exact system. These two functions are called `fmap` and `join`. Below, I'll implement them for the Option Monad.

```
def fmap(self, function):
    if self.is_none():
        return self
    val = self.unwrap()
    return Option.some(function(val))

def join(self):
    if self.is_none():
        return self
    val = self.unwrap()
    return val
```

`fmap` is the simpler of the two. `fmap` applies a function to the Option's value, if it has one. The difference between `fmap` and `bind` is that `fmap` assumes that it's function will always succeed, and therefore doesn't need to return an Option Monad.

```
def plus_one(x):
    return x + 1

Option.some(5).fmap(plus_one)
```

In the above example, we can't use `bind`, because `plus_one` doesn't return an Option Monad, so it would break our chain of `bind` commands. However, `fmap` can be used instead. But `fmap` can't be a replacement for `bind` all on it's own.

```
def one_over(x):
    if x == 0:
        return Option.none()
    return Option.some(1/x)

Option.some(5).fmap(one_over) == Option.some(Option.some(0.2))
Option.some(0).fmap(one_over) == Option.some(Option.none())
```

In the above example, `fmap` adds an additional layer of the Option Monad. This is where `join` comes in.

```
Option.some(5).fmap(one_over).join() == Option.some(0.2)
Option.some(0).fmap(one_over).join() == Option.none()
Option.none().fmap(one_over).join() == Option.none()
```

As a matter of practice, most monads will implement all three; `bind`, `fmap`, and `join`. Annoyingly, different languages and libraries name these functions different things, but they're probably there under different names.

5 The Zeroth Monad

I still have to write this part. It's about lists. Yeah, lists are monads.